Algorithmic specifications: A new specification method for abstract data types (*)

by

Jacques Loeckx

A 81/14

(submitted for publication)

December 1981

Fachbereich 10 - Informatik
Universität des Saarlandes
6600 Saarbrücken
West Germany

(*) The present paper is a completely revised and extended version of a paper presented at the 8^{th} ICALP Conference [20].

1. Introduction

Algebraic specifications of abstract data types have been discussed in a large number of papers [30, 10, 12, 2, 15, 17]. While being based on a attractive idea they put a series of difficult theoretical and practical problems such as the error problem [11,14], the partial function problem [2], the problems of consistency and sufficiently-completeness [12] and the problem of enrichment and extension [10, 7, 3]; moreover, writing specifications for a given data type is not necessarily a trivial exercise - as is illustrated by the data type *Set-of-Integers* of [12,14]; one of the axioms for the function *Delete* removing an element from a set is:

$$\begin{aligned} &\mathrm{Delete}(\mathrm{Insert}(s, i), j) = \\ &\quad \underline{\mathrm{if}}\ i = j\ \underline{\mathrm{then}}\ \mathrm{Delete}(s, j) \\ &\qquad\qquad\ \underline{\mathrm{else}}\ \mathrm{Insert}(\mathrm{Delete}(s, j), i) \quad ; \end{aligned}$$

the then-clause of this equation is Delete(s, j) rather than s because an element of the data type *Set-of-Integers* may contain duplicates; intuitively it is not directly clear why these duplicates may occur nor where they occur.

The present paper proposes an alternative specification method avoiding these different problems. Using continuous functions on complete partial orders it solves the problem of partial functions; defining the functions algorithmically it avoids the problems of consistency and sufficiently-completeness; using a specification to deduce a new algebra from a given one it avoids the extension problem; finally, using a formal language - or, more precisely, the flat complete partial order corresponding to this language - as a domain for the functions introduced it provides a clear answer to questions of the kind illustrated above.

Our concern in this paper is twofold. First, we want our specification method to have a sound theoretical basis. Second, we want it to constitute a practical tool for the top-down development of (modularly structured) programs and for the (modular) verification of these programs with a verification system in the style of, say, AFFIRM [25].

Section 2 recalls some notions concerning continuous algebras. The principle of the algorithmic specification method is shortly described in Section 3. The detailed description of this method is in Sections 4 and 5; at the end of Section 5 a more convenient notation is introduced. Section 6 is concerned with the verification of specifications. Properties of abstract data types and their proofs are discussed in Section 7. The use of simultaneous fixpoint abstraction is discussed in Section 8. Parameterized data types are introduced in Section 9. Section 10 presents some conclusions including a comparison with similar work.

2. Continuous algebras

2.1 Algebras and continuous algebras

Let $\underline{S}$ be a set of *types* (or: *sorts*).

An $\underline{S}$-*typed signature* $\underline{\Sigma}$ is the union of disjoint sets $\underline{\Sigma}_{\sigma_1\sigma_2\ldots\sigma_n,\sigma}$ with $\sigma_1,\ldots,\sigma_n$, $\sigma \in \underline{S}$, $n \geq 0$. An element $f \in \underline{\Sigma}_{\sigma_1\ldots\sigma_n,\sigma}$ is called a *function symbol* of type σ. Instead of writing

$$f \in \underline{\Sigma}_{\sigma_1\ldots\sigma_n,\sigma}$$

we write

$$f : \sigma_1 \times \sigma_2 \times\ldots\times \sigma_n \to \sigma \qquad \text{if } n \geq 1$$

and

$$f : \to \sigma \qquad \text{if } n = 0$$

Let $\underline{\Sigma}$ be an S-typed signature. A (*heterogeneous*) $\underline{\Sigma}$-*algebra* A consists of:

(i) a set $\underline{C}_\sigma$ for each type $\sigma \in \underline{S}$, called the *carrier* of A of type σ;

(ii) a function

$$f^{(A)} : \underline{C}_{\sigma_1} \times \ldots \times \underline{C}_{\sigma_n} \to \underline{C}_\sigma$$

for each function symbol

$$f : \sigma_1 \times \ldots \times \sigma_n \to \sigma$$

of $\underline{\Sigma}$.

A $\underline{\Sigma}$-algebra is called *continuous* if its carrier sets are complete partial orders (c.p.o.'s) and if its functions are continuous (with respect to these c.p.o.'s) (cf. [9]).

Note that a continuous algebra may be considered as an applicative programming language; a program of this language is a (correctly typed) expression built up from function symbols of the algebra.

2.2 Standard algebras

A continuous algebra is called *standard* if it satisfies the following three conditions:

(i) its carrier sets are flat c.p.o.'s; $=_\sigma$, $\sqsubseteq_\sigma$ and $\perp_\sigma$ (or, if no ambiguity arises, $=$, $\sqsubseteq$ and $\perp$) denote respectively the equality, the partial order and the bottom element of the carrier set $\underline{C}_\sigma$ (see Figure I);

(ii) it contains a carrier set $\underline{C}_{Bool} = \{\underline{true}, \underline{false}, \perp_{Bool}\}$ of type *Bool*;

(iii) for each of its carrier sets $\underline{C}_\sigma$ it contains a strict [1] function

$$\underline{C}_\sigma^{\,2} \to \underline{C}_{Bool}$$

expressing the equality in $\underline{C}_\sigma - \{\perp_\sigma\}$; this function is denoted by the infix operator "$\doteq_\sigma$" or, shortly, "$\doteq$".

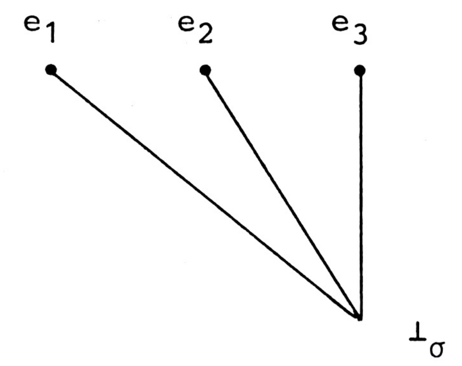

FIGURE I: The flat c.p.o. $(\underline{C}_\sigma, \sqsubseteq_\sigma)$ with $\underline{C}_\sigma = \{e_1, e_2, \ldots\} \cup \{\perp_\sigma\}$

---

[1] A function is *strict* if its value is the bottom element as soon as one of its argument is a bottom element.

It is important to distinguish between "$=_\sigma$" and "$\dot{=}_\sigma$": the latter is a (strict) predicate (i.e. function) of the algebra; the former is a (non-continuous) predicate about the algebra. For instance:

$$\perp_\sigma =_\sigma \perp_\sigma$$

but

$$(\perp_\sigma \dot{=}_\sigma \perp_\sigma) = \perp_{Bool}$$

(with "=" standing for "$=_{Bool}$").

In the sequel only standard continuous algebras are considered.

2.3 Enriching a continuous algebra by additional functions

A (new) continuous function may be defined as an expression which is built up from variables and (existing) continuous functions by composition, λ-abstraction and minimal fixpoint abstraction. Following the LCF-notation [23] minimal fixpoint abstraction is expressed with the help of the operator α : if e is an expression and M a function variable, [αM.e] denotes the minimal fixpoint of [λM.e].

A typical definition is that of the function Factorial:

$$\text{Factorial} = [\alpha M.\ [\lambda n \in \underline{C}_{\text{Integer}}.\ \underline{\text{if}}\ n \dot{=} 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ n \times M(n-1)]] \qquad (1)$$

For more details on the theoretical background and on the notation the reader is referred to [23, 24]. Note in particular that the equality "=" in the expression (1) may be considered as an extension of "$=_\sigma$" and is therefore not continuous: a definition such as

$$\text{Factorial} = e$$

stands for

$$\text{for all } i \in \underline{C}_{\text{Integer}} : \text{Factorial}(i) =_{\text{Integer}} e(i).$$

3. The principles of the algorithmic specification method

The goal of an algorithmic specification is to extend a given algebra with a new data type.

Let A be a $\underline{\Sigma}$-algebra with $\underline{\Sigma}$ an $\underline{S}$-typed signature and $\underline{S}$ a set of types. The goal of a specification of a type σ, $\sigma \notin \underline{S}$, is to add a type σ to the $\underline{\Sigma}$-algebra A; more precisely, the specification defines (together with the algebra A):

- the set of types $\underline{S}_\sigma = \underline{S} \cup \{\sigma\}$;
- a $\underline{S}_\sigma$-typed signature $\underline{\Sigma}_\sigma$ obtained from $\underline{\Sigma}$ by adding some function symbols;
- a $\underline{\Sigma}_\sigma$-algebra A_σ obtained by adding to A a carrier set $\underline{C}_\sigma$ and a function for each function symbol added; A_σ is a standard continuous algebra if A is.

Essentially a specification of type σ consists of a set of "constructors" and a set of functions called "user functions".

The goal of the constructors is to define a formal language called "term language". Syntactically, a constructor is a function symbol (of type σ); semantically constructors are merely building blocks of a formal language, - contrasting with the classical interpretation of function symbols as functions.

The user functions are functions defined in a $\underline{\Sigma}_\sigma$-algebra $A_\sigma^{(L)}$ along the lines of Section 2.3; this algebra $A_\sigma^{(L)}$ is obtained by adding to A the term language (defined by the constructors) as the carrier of type σ. Among the user functions is a special function defining an equivalence relation on the term language.

Finally, considering an homomorphism mapping the term language into its equivalence classes, one may - roughly speaking - define the algebra A_σ as the homomorphic image of the algebra $A_\sigma^{(L)}$ enriched by the user functions.

These different notions will now be explained more precisely in the Sections 4 to 6.

While assuming that the (continuous) algebra A is standard we leave open the question whether the types of A - except *Bool* - are ground types (i.e. "given" types) or have been introduced by previous specifications.

4. The algebra $A_\sigma^{(L)}$

Let A be a $\underline{\Sigma}$-algebra with $\underline{\Sigma}$ an $\underline{S}$-typed signature.

Let σ be a type, $\sigma \notin \underline{S}$; put $\underline{S}_\sigma = \underline{S} \cup \{\sigma\}$.

The goal of this Section is to define the algebra $A_\sigma^{(L)}$ associated with the algebra A and the type σ.

4.1 Constructor sets

A *constructor set (of the type* σ *for the algebra A)* is a set C[A,σ] of function symbols

$$f : \sigma_1 \times \ldots \times \sigma_n \to \sigma$$

with $n \geq 0$ and $\sigma_1, \ldots, \sigma_n \in \underline{S}_\sigma$.

As an example the constructor set of the type *Set* for an algebra containing the type *Integer* may consist of the constructors:

emptyset : $\to$ *Set*
insert : *Set* x *Integer* $\to$ *Set*

(see Figure V)

In order to distinguish constructors from (other) function symbols we adopt the convention that - in the examples - the former start with a lower-case letter, the latter with a higher-case letter.

4.2 Term languages

Consider a constructor set $C[A,\sigma]$.

Call $\underline{M}_{A,\sigma}$ the formal language inductively defined by:

(i) if $f : \to \sigma$ is a constructor, then $f \in \underline{M}_{A,\sigma}$

(ii) if $f : \sigma_1 \times \ldots \times \sigma_n \to \sigma$, $n \geq 1$, is a constructor and if for all i, $1 \leq i \leq n$,

$$s_i \in \underline{M}_{A,\sigma} \text{ when } \sigma_i = \sigma$$

and

$$s_i \in \underline{C}_{\sigma_i} - \{\perp_{\sigma_i}\} \text{ when } \sigma_i \neq \sigma \quad [2]$$

then

$$f(s_1, \ldots, s_n) \in \underline{M}_{A,\sigma}$$

The *term language* (of the constructor set $C[A,\sigma]$) is the flat c.p.o. $\underline{L}_{A,\sigma}$ - or, shortly, $\underline{L}_\sigma$ - obtained by adding to $\underline{M}_{A,\sigma}$ a bottom element, say $\perp_\sigma^{(L)}$, and providing it with a partial order, say $\sqsubseteq_\sigma^{(L)}$. (see Figure II). The elements of a term language are called *terms (of type σ)*. Examples of terms are (cf. Figure V):

emptyset

insert (insert(emptyset, 3), 1)

$\perp_{Set}^{(L)}$

but <u>not</u>

insert ($\perp_{Set}^{(L)}$, 1)

or insert (emptyset, $\perp_{Integer}$)

4.3 Term functions

We now associate with the following (continuous) functions, called the *term functions* (of $C[A,\sigma]$):

---

[2] s_i has, strictly speaking, to be defined as a representation of an element of $\underline{C}_{\sigma_i}$, not as the element itself.

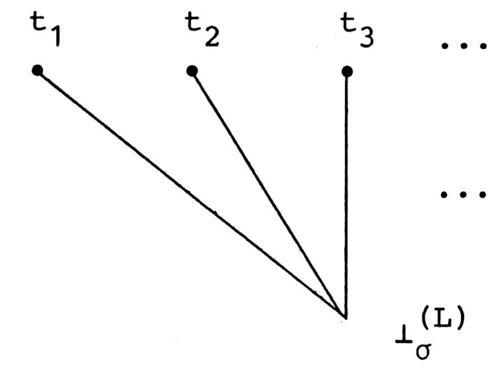

FIGURE II: The flat c.p.o. $(\underline{L}_\sigma, \underline{\sqsubseteq}_\sigma^{(L)})$ associated with the language $\underline{M}_{A,\sigma} = \{t_1, t_2, \ldots\}$

(i) the strict function

$$\text{Equal-}\sigma : \underline{L}_\sigma^2 \to \underline{C}_{Bool}$$

which expresses the syntactical equality of terms, i. e. the equality in the formal language $\underline{L}_\sigma - \{\perp_\sigma\}$;

(ii) the function

If-then-else$_\sigma$: $C_{Bool} \times \underline{L}_\sigma \times \underline{L}_\sigma \to \underline{L}_\sigma$:

$$\text{If-then-else}_\sigma(b, s, t) = \begin{cases} s & \text{if } b = \underline{\text{true}} \\ t & \text{if } b = \underline{\text{false}} \\ \perp_\sigma^{(L)} & \text{if } b = \perp_{Bool} \end{cases} ;$$

(iii) for each constructor $f : \sigma_1 \times \ldots \times \sigma_n \to \sigma$, $n \geq 0$, the strict function

$$\text{Is-f} : \underline{L}_\sigma \to \underline{C}_{Bool}$$

which expresses that the leftmost symbol of its argument is f ;

(iv) for each constructor $f : \sigma_1 \times \ldots \times \sigma_n \to \sigma$, $n \geq 1$, the strict function

$$\text{Cons-f} : \underline{K}_1 \times \ldots \times \underline{K}_n \to \underline{L}_\sigma:$$

$$\text{Cons-f}(s_1,\dots,s_n) = f(s_1,\dots,s_n)$$ [3]

with, for each i, $1 \le i \le n$: $\underline{K}_i = \begin{cases} \underline{L}_{\sigma_i} & \text{if } \sigma_i = \sigma \\ \underline{C}_{\sigma_i} & \text{if } \sigma_i \neq \sigma \end{cases}$

which constructs a term;

(v) for each constructor $f : \sigma_1 \times \dots \times \sigma_n \to \sigma$, $n \ge 1$, and each i, $1 \le i \le n$, the strict function

$$\text{Arg}_i\text{-f} : \underline{L}_\sigma \to \underline{K}_i :$$

$$\text{Arg}_i\text{-f}(t) = \begin{cases} t_i & \text{if } t \text{ has the form } f(t_1,\dots,t_n) \\ \perp^{(L)}_{\sigma_i} & \text{else} \end{cases}$$ [4]

$$\text{with } \underline{K}_i = \begin{cases} \underline{L}_{\sigma_i} & \text{if } \sigma_i = \sigma \\ \underline{C}_{\sigma_i} & \text{if } \sigma_i \neq \sigma \end{cases}$$

which extracts an "argument" of a term.

---

[3] Strictly speaking, $\text{Cons-f}(s_1,\dots,s_n) = f(\tilde{s}_1,\dots,\tilde{s}_n)$ where $\tilde{s}_i = \begin{cases} s_i & \text{if } \sigma_i = \sigma \\ \text{a representation of } s_i & \text{if } \sigma_i \neq \sigma. \end{cases}$

[4] Strictly speaking, if t has the form $f(t_1,\dots,t_n)$ then $\text{Arg}_i\text{-f}(t) = \begin{cases} t_i & \text{if } \sigma_i = \sigma \\ \text{the element represented by } t_i & \text{if } \sigma_i \neq \sigma \end{cases}$

4.4 The algebra $A_\sigma^{(L)}$

The algebra $A_\sigma^{(L)}$ corresponding to a constructor set C[A,σ] is defined as follows:

- its carrier sets are those of A together with a carrier set for σ, viz. the term language $\underline{L}_\sigma$ of C[A,σ];
- its functions are those of A together with the term functions of C[A,σ].

Informally, $A_\sigma^{(L)}$ is an algebra A extended with a type σ; the objects of type σ are terms, i. e. words of a formal language.

4.5 Using structural induction

The principle of structural induction ([4, 1]) being applicable on the term language $\underline{L}_\sigma$, it may be used in proofs properties of the algebra $A_\sigma^{(L)}$.

As an example suppose one has the prove:

for all $s \in \underline{L}_\sigma$, $t \in \underline{C}_\tau$: $q(s, t)$ holds (1)

where $\tau \neq \sigma$. Suppose moreover the constructor set C[A,σ] consists of:

$$f_0 : \rightarrow \sigma$$

$$f_1 : \sigma \times \rho \rightarrow \sigma \qquad \text{with } \rho \neq \sigma$$

For proving (1) it is sufficient to prove:

(i) for all $t \in \underline{C}_\tau$: $q(\perp_\sigma^{(L)}, t)$ holds

(ii) for all $t \in \underline{C}_\tau$: $q(f_0, t)$ holds

(iii) for all $s \in \underline{L}_\sigma$, $t \in \underline{C}_\tau$, $r \in \underline{C}_\rho$:

if for all $t' \in \underline{C}_\tau$ $q(s, t')$ holds
then $q(f_1(s, r), t)$ holds

5. Algorithmic specifications

Let $\underline{S}$, $\underline{\Sigma}$, A, σ, C[A,σ], $\underline{L}_\sigma$, $A_\sigma^{(L)}$ be defined as in Section 4. We now indicate the form of a specification of the type σ; we then describe how this specification defines the algebra A_σ.

5.1 Specifications

An algorithmic specification of the type σ (for the algebra A) consists of six sections defining respectively (cf. Figure III)

(i) the name of the type σ and the names of types which are assumed to be in $\underline{S}$; these latter types are called the *underlying types*;

(ii) the constructor set C[A,σ];

(iii) a function Is.σ called *acceptor function* [5];

(iv) a function Eq.σ called *equivalence relation* [5];

(v) a set of functions called *external functions*;

(vi) a (possibly empty) set of functions called *auxiliary* (or: *hidden*) *functions*.

The acceptor function is a strict function

$$\text{Is.}\sigma : \underline{L}_\sigma \to \underline{C}_{\text{Bool}}$$

We put

$$\underline{L}'_\sigma = \{s \in \underline{L}_\sigma \mid \text{Is.}\sigma(s) = \underline{\text{true}}\} \cup \{\bot_\sigma^{(L)}\}$$

and call the formal language $\underline{L}'_\sigma - \{\bot_\sigma^{(L)}\}$ the *subset of the term language defined by the acceptor function.*

The equivalence relation is a strict function

$$\text{Eq.}\sigma : \underline{L}_\sigma^{\,2} \to \underline{C}_{\text{Bool}}$$

---

[5] Do not confuse Is.σ and Eq.σ with a term functions Is-f and Equal-σ.

(i) General

Type : *Stack*

Underlying types : *Integer, Bool*

(ii) Constructors

emptystack : $\to$ *Stack*

push : *Stack* x *Integer* $\to$ *Stack*

(iii) Acceptor function

$$\text{Is.Stack} = [\lambda s \in \underline{L}_{\text{Stack}}.\ \underline{\text{if}}\ \text{Depth}(s) \leq 10\ \underline{\text{then}}\ \underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{false}}]$$

(iv) Equivalence relation

Eq.Stack = Equal-Stack

(v) External functions

Emptystack = emptystack

$$\text{Push} = [\lambda s \in \underline{L}_{\text{Stack}}.\ [\ \lambda i \in \underline{C}_{\text{Integer}}.\ \underline{\text{if}}\ \text{Depth}\ (s) < 10\ \underline{\text{then}}\ \text{Cons-push}\ (s,i)\ \underline{\text{else}}\ \bot^{(L)}_{\text{Stack}}\ ]]$$

$$\text{Pop} = [\lambda s \in \underline{L}_{\text{Stack}}.\ \underline{\text{if}}\ \text{Is-push}\ (s)\ \underline{\text{then}}\ \text{Arg}_1\text{-push}\ (s)\ \underline{\text{else}}\ \bot^{(L)}_{\text{Stack}}]$$

$$\text{Top} = [\lambda s \in \underline{L}_{\text{Stack}}.\ \underline{\text{if}}\ \text{Is-push}\ (s)\ \underline{\text{then}}\ \text{Arg}_2\text{-push}\ (s)\ \underline{\text{else}}\ \bot_{\text{Integer}}]$$

$$\text{Isnew} = [\lambda s \in \underline{L}_{\text{Stack}}.\ \underline{\text{if}}\ \text{Is-push}(s)\ \underline{\text{then}}\ \underline{\text{false}}\ \underline{\text{else}}\ \underline{\text{true}}]$$

(vi) Auxiliary function

$$\text{Depth} = [\alpha M.[\lambda s \in \underline{L}_{\text{Stack}}.\ \underline{\text{if}}\ \text{Is-push}\ (s)\ \underline{\text{then}}\ M(\text{Arg}_1\text{-push}(s)) + 1\ \underline{\text{else}}\ 0\ ]]$$

<u>FIGURE III:</u> A specification of the data type *Stack* for an algebra containing (at least) the type *Integer* (and *Bool*). Intuitively, the data type consists of stacks of integers with a maximal depth of 10.

such that its restriction

$$\mathrm{Eq}.\sigma \mid (\underline{L}'_\sigma - \{\bot_\sigma^{(L)}\})^2$$

is an equivalence relation in the language defined by the acceptor function i. e. in the formal language $\underline{L}'_\sigma - \{\bot_\sigma^{(L)}\}$.

Finally the external and auxiliary functions take arguments and values in the term language $\underline{L}_\sigma$ and in the carrier sets of the underlying types.

Each function of the specification is defined as a function in the algebra $A_\sigma^{(L)}$ possibly already enriched by some (other) functions of the specification. More precisely, the right-hand side of a function definition may contain functions of $A_\sigma^{(L)}$ (i. e. functions of A and/or term functions) as well as the functions being defined in the specification, provided there exists no sequence

$$F_1, F_2, \ldots, F_m \qquad , m \geq 2 \qquad \text{(C1)}$$

of functions being defined in the specification with $F_m = F_1$ and F_{i+1} occurring in the right-hand side of) the definition of F_i, $1 \leq i \leq m - 1$.[6]

The equivalence relation, the external functions and the term function If-then-else$_\sigma$ are called the *user functions (introduced by the specification of type σ).*

Note that the external or auxiliary functions are not required to have at least one argument or the value ranging over $\underline{L}_\sigma$: in the specification of Figure III it is for instance possible to introduce a function mapping integers into integers.

[6] This condition will be relaxed in Section 8.

5.2 The algebra defined

If s is an element of a carrier set of the algebra $A_\sigma^{(L)}$ let $\varphi(s)$ denote:

- s, if $s \in \underline{C}_\tau$ for any type $\tau \neq \sigma$;
- the equivalence class of the term s induced on the formal language $\underline{L}'_\sigma - \{\perp_\sigma^{(L)}\}$ by the equivalence relation

  Eq.σ, if $s \in \underline{L}'_\sigma - \{\perp_\sigma^{(L)}\}$;

- an element denoted $\perp_\sigma$, if $s = \perp_\sigma^{(L)}$.

Note that the notation $\varphi(s)$ is left undefined for $s \in \underline{L}_\sigma - \underline{L}'_\sigma$.

We are now able to define the $\underline{\Sigma}_\sigma$-algebra A_σ defined by the $\underline{\Sigma}$-algebra A and the specification of type σ.

The signature $\underline{\Sigma}_\sigma$ is obtained by adding to $\underline{\Sigma}$ a function symbol for each user function. Informally, a specification contributes to the new algebra only by its user functions, not by its auxiliary functions (nor by its term functions other than If-then-else$_\sigma$).

The carrier sets of the algebra A_σ are the carriers sets of the algebra A together with the carrier set $\underline{C}_\sigma$ of type σ defined by

$$\underline{C}_\sigma = \{\varphi(s) \mid s \in \underline{L}'_\sigma\}$$

$\underline{C}_\sigma$ is defined as a flat c.p.o. with $\perp_\sigma$ $(= \varphi(\perp_\sigma^{(L)}))$ as its minimal element. Informally, the carrier set $\underline{C}_\sigma$ consists of the equivalence classes induced by Eq.σ on the subset of $\underline{L}_\sigma$ defined by the acceptor function.

Finally, the functions of the algebra A_σ are the functions of the algebra A together with a function $F^{(A)}$ for each user function F of the specification; more precisely, if F is an n-ary function with arguments of type $\sigma_1,\ldots,\sigma_n$ and values of type σ_{n+1}, $n \geq 0$, then

$$F^{(A)} : \underline{C}_{\sigma_1} \times \ldots \times \underline{C}_{\sigma_n} \rightarrow \underline{C}_{\sigma_{n+1}} :$$

$$F^{(A)}(\varphi(s_1), \ldots , \varphi(s_n)) = \varphi(F(s_1, \ldots , s_n))$$

Informally, $F^{(A)}$ is the image of F under the homomorphism φ mapping terms into their equivalence classes.

Note that for the definition of $F^{(A)}$ to be consistent the function F must satisfy certain conditions. Informally speaking, F has to preserve the predicates Is.σ and Eq.σ: the first condition guarantees the existence[7] of the value of $F^{(A)}$ by checking that $F(s_1,\dots,s_n) \in \underline{L}'_\sigma$ whenever F has values in $\underline{L}_\sigma$; the second condition guarantees the uniqueness of the value of $F^{(A)}$. The study of these conditions is the subject of Section 6.

Note that $(\mathrm{Eq}.\sigma)^{(A)}$ is the (strict function expressing the) equality in the carrier set $\underline{C}_\sigma$, i.e. the function denoted by the infix operator "$\doteq_\sigma$".

It is easy to show that A_σ is again a standard continuous algebra.

5.3 A few informal comments

The purpose of the acceptor function is to eliminate some terms from consideration. For instance, in the data type *Stack* of Figure III the attention is restricted to terms containing not more than 10 (stacked) integers; in the data type *Set* of Figure V the acceptor function eliminates the terms with duplicates.[8]

The purpose of the equivalence relation is to "identify" terms which are syntactically different. In the data type *Stack* there is a one-to-one correspondence between the terms and the "stacks" of the carrier set; in the data type *Set* an element of the carrier set corresponds to all terms differing only by the order

---

[7] Remember that $\varphi(s)$ is not defined for $s \in \underline{L}_\sigma - \underline{L}'_\sigma$.

[8] The notation used in Figure V will be introduced in Section 5.4.

of occurrence of the integers.

In general there exist several possible algorithmic specifications for a given data type which are more or less "natural". These specifications differ by the choice for the constructors and the acceptor function. For instance we may define Is.Set in Figure V as the strict extension of the function with the constant value true; modifying the definition of Eq.Set and of the external functions accordingly leads to a specification with duplicates defining the same data type *Set* (up to isomorphism).

In the example of Figure III "errors" such as stack overflow or the "popping" of an empty stack lead to the undefined value "⊥". The use of this value has the advantage that it is automatically "transmitted" by any (strict) function; as a drawback it does not allow branchings because of the monotonicity of the functions: the expression

$$\underline{\text{if}}\ s \doteq \bot\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ e_2$$

for instance, delivers the value ⊥, not e_1 or e_2. Instead of using "⊥" it is also possible to use a special term, say "errorstack" - or even several terms such as "stackoverflow" and "poppingerror". In that case the constructor set has to be augmented correspondingly, for instance by

errorstack: → *Stack*

More importantly, the function definitions have then to take into account the additional case "Is-errorstack(s)"; this leads to the annoying obligation to explicitly specify the error transmission through all underlying data types, for instance by defining the value of Isnew for the argument errorstack.

5.4 A more appealing notation for function definitions

Our concern to be precise has led us to introduce a heavy formalism - as illustrated by Figure III. We now introduce a notation which usually allows to avoid the explicit use of term functions such as Arg_i-f and the use of function variables required by the α-notation.

The notation is defined by the following rules which are illustrated by examples rather than being defined for the general case:

(i) (Elimination of λ and explicit indication of the type of the function values):

A definition such as

$$F(s : \sigma, t : \tau) : \rho = e$$

stands for

$$F = [\lambda s \in \underline{L}_{\sigma} . [\lambda t \in \underline{C}_{\tau} . e]]$$

provided ρ is the type of the values of F and e contains no occurrences of F. Attention: do not forget the main difference between σ and the other types: s ranges over $\underline{L}_{\sigma}$, t over $\underline{C}_{\tau}$.

(ii) (Elimination of α):

A definition such as

$$F (s : \sigma, t : \tau) : \rho = e$$

stands for

$$F = [\alpha M. [\lambda s \in \underline{L}_{\sigma} . [\lambda t \in \underline{C}_{\tau} . e_F^M]]]$$

provided ρ is the type of the values of F, e contains at least one occurrence of F and e_F^M is the result of substituting M for F in e. Attention: do not forget that the equality "hides" a minimal fixpoint abstraction.

(iii) (Elimination of the term function Cons-f):

An expression such as

$$f(e_1,\dots,e_n)$$

where $e_1, \dots , e_n$ are expressions, stands for

$$\text{Cons-f } (e_1,\dots,e_n)$$

Attention: $f(\dots, \bot, \dots)$ stands for $\bot$, <u>not</u> for the word $f(\dots, \bot, \dots)$.

(iv) (Elimination of the term functions Is-f and Arg_i-f by the use of the case construction and the introduction of additional variables)

Assume the constructor set $C[A,\sigma]$ consists of:

$$f_0 : \to \sigma$$

$$f_1 : \sigma \times \rho \to \sigma \qquad \text{with } \rho \neq \sigma$$

$$f_2 : \sigma \times \sigma \to \sigma$$

```
(i)    General
        Type : Stack
        Underlying types: Integer, Bool
(ii)   Constructors
              emptystack : → Stack
              push : Stack x Integer → Stack
(iii)  Acceptor function
           Is.Stack (s : Stack) : Bool
             = if Depth(s) ≤ 10 then true else false
(iv)   Equivalence relation
           Eq.Stack (s1 : Stack, s2 : Stack) : Bool
                     = Equal-Stack (s1, s2)
(v)    External functions
           Emptystack ( ) : Stack = emptystack
           Push (s : Stack, i : Integer) : Stack
             = if Depth(s) < 10 then push(s, i) else ⊥
           Pop (s : Stack) : Stack
             = case
                 s = emptystack : ⊥
                 s = push (s', i) : s'
               esac
           Top (s : Stack) : Integer
             = case
                 s = emptystack : ⊥
                 s = push (s', i) : i
               esac
           Isnew (s : Stack) : Bool
             = case
                 s = emptystack : true
                 s = push (s', i) : false
               esac
(vi)    Auxiliary function
           Depth (s : Stack) : Integer
             = case
                 s = emptystack : 0
                 s = push (s', i) : Depth (s') + 1
               esac
```

Figure IV: The specification of Figure III in the notation of Section 5.4.

An expression such as

case
$s = f_0 : e_0$
$s = f_1(s', r) : e_1$
$s = f_2(s'', s''') : e_2$
esac

stands for

if $\text{Is-}f_0(s)$ **then** e'_0
else **if** $\text{Is-}f_1(s)$ **then** e'_1
else **if** $\text{Is-}f_2(s)$ **then** e'_2
else $\perp$

where

- s', s'', s''', r are "new" variables;
- e'_1 is obtained from e_1 by replacing the occurrences of s' by $\text{Arg}_1\text{-}f_1(s)$ and r by $\text{Arg}_2\text{-}f_1(s)$;
- e'_2 is obtained from e_2 in a similar way.

As an example the reader may compare the Figures III and IV; see also Figure V.

```
(i)   General
        Type : Set
        Underlying types : Integer, Bool
(ii)  Constructors
        emptyset : Set
        insert : Set x Integer → Set
(iii) Acceptor function
        Is.Set(s : Set) : Bool
         = case
             s = emptyset : true
             s = insert (s', i) :
                if Memberof (s', i) then false
                                    else Is.Set (s')
           esac
(iv)  Equivalence relation
        Eq.Set (s1 : Set, s2 : Set) : Bool
         = if Subset (s1, s2) then Subset (s2, s1) else false
(v)   External functions
        Emptyset( ) : Set = emptyset
        Insert (s : Set, i : Integer) : Set
         = if Memberof (s, i) then s else insert (s, i)
        Delete (s : Set, i : Integer) : Set
         = case
             s = emptyset : s
             s = insert (s', i') :
                if i' ≐ i then s' else insert (Delete(s', i), i')
           esac
        Memberof (s : Set, i : Integer) :  Bool
         = case
             s = emptyset : false
             s = insert (s', i') :
                if i' ≐ i then true else Memberof (s', i)
           esac
        Subset (s1 : Set, s2 : Set) : Bool
         = case
             s1 = emptyset : true
             s1 = push (s1', i) :
                 if Memberof (s2, i) then Subset (s1', s2)
                                     else false
           esac
```

FIGURE V: A specification of the data type *Set*. Note that Is.Set avoids the occurrence of duplicates in the term language and that Eq.Set identifies terms which differ only by the order of occurrence of their elements.

6. The verification of specifications

6.1 Preliminary remark

The verification of a specification consists in verifying for it the consistency of the definition of the algebra A_σ (see Section 5.2). The verification of a specification does not include a proof of the syntactical correctness which should, among other things, make sure that the right-hand sides of the function definitions are correctly typed. It is also different from a (semantical) "correctness proof" checking that the data type defined corresponds to the "intended" one - whatever this means.

6.2 The verification conditions

The definition of the algebra A_σ defined by a specification of type σ for the algebra A is consistent if the specification satisfies the following conditions:

(i) Is.σ and Eq.σ are strict functions;

(ii) Eq.σ is an equivalence relation in $\underline{L}'_\sigma$, i.e. Eq.σ is a total, reflexive, symmetric and transitive relation; this condition has to be verified because the definition of Eq.σ merely guarantees that it is a (possibly partial) function with values of type *Bool*;

(iii) each user function with values in $\underline{L}_\sigma$ preserves the property Is.σ, i.e. the function value satisfies Is.σ if the arguments do;

(iv) each user function preserves the equivalence relation Eq.σ, i. e. equivalent arguments lead to equivalent values.

More precisely, the *verification conditions* of the specification of data type σ are:

(i) (a) Is.$\sigma(\perp) = \perp$

(b) For all $s \in \underline{L}_\sigma$:

if Is.$\sigma(s)$ = true

then Eq.$\sigma\ (s,\perp)$ = Eq.$\sigma(\perp, s) = \perp$ [9]

[9] Eq.$\sigma(\perp,\perp) = \perp$ has not to be checked because it results from the condition (i) (b) by monotonicity.

(ii) For all $s, s_1, s_2, s_3 \in \underline{L}_\sigma$:

if $Is.\sigma(s) = Is.\sigma(s_1) = Is.\sigma(s_2) = Is.\sigma(s_3) = \underline{true}$

then

(a) either $Eq.\sigma\ (s_1, s_2) = \underline{true}$ or $Eq.\sigma(s_1, s_2) = \underline{false}$

(b) $Eq.\sigma(s, s) = \underline{true}$

(c) $Eq.\sigma(s_1, s_2) = Eq.\sigma(s_2, s_1)$

(d) if $Eq.\sigma(s_1, s_2) = Eq.\sigma(s_2, s_3) = \underline{true}$

then $Eq.\sigma(s_1, s_3) = \underline{true}$

(iii) For each external function

$$F(s_1 : \sigma_1, \ldots, s_n : \sigma_n) : \sigma \quad , n \geq 0$$

(with values in $\underline{L}_\sigma$) one has:

<u>for all</u> $s_i \in \underline{L}_\sigma$ with either $Is.\sigma(s_i) = \underline{true}$ or $s_i = \perp$, *if* $\sigma_i = \sigma$, $1 \leq i \leq n$,

<u>and for all</u> $s_i \in \underline{C}_{\sigma_i}$, *if* $\sigma_i \neq \sigma$, $1 \leq i \leq n$:

$$\begin{cases} \text{either } Is.\sigma(F(s_1,\ldots,s_n)) = \underline{true} \\ \text{or } F(s_1,\ldots,s_n) = \perp \end{cases}$$

(iv) For each external function

$$F(s_1 : \sigma_1, \ldots, s_n : \sigma_n) : \sigma_{n+1} \quad , \quad n \geq 0$$

one has:

<u>for all</u> $s_i, s'_i \in \underline{L}_\sigma$ with $\begin{cases} \text{either } Is.\sigma(s_i) = Is.\sigma(s'_i) = \underline{true} \\ \quad \text{and } Eq.\sigma(s_i, s'_i) = \underline{true} \\ \text{or } s_i = s'_i = \perp, \end{cases}$

if $\sigma_i = \sigma$, $1 \leq i \leq n$

<u>and for all</u> $s_i, s'_i \in \underline{C}_{\sigma_i}$ with $s_i = s'_i$, *if* $\sigma_i \neq \sigma$, $1 \leq i \leq n$:

$$\begin{cases} \text{either } Eq.\sigma(F(s_1,\ldots,s_n), F(s'_1,\ldots,s'_n)) = \underline{true} \\ \text{or } F(s_1,\ldots,s_n) = F(s'_1,\ldots,s'_n) = \perp \end{cases} \quad \textit{if } \sigma_{n+1} = \sigma$$

and

$$F(s_1,\ldots,s_n) = F(s'_1,\ldots,s'_n) \quad , \textit{if } \sigma_{n+1} = \sigma$$

Note that for all types σ the user function If-then-else$_\sigma$ satisfies the conditions (iii) and (iv) and has therefore not to be checked; a similar remark holds for the equivalence relation $Eq.\sigma$ and the condition (iv).

It is interesting to note that these verification conditions are very similar to those of [13].

As an example the verification conditions (iii) and (iv) for the data type *Set* of Figure V are:

(iii) for all $s \in \underline{L}_{Set}$ and all $i \in \underline{C}_{Integer}$:

if: Is.Set(s) = true or $s = \perp$

then: (a) Is.Set(Insert(s, i)) = true or Insert (s, i) = $\perp$

(b) Is.Set(Delete(s, i)) = true or Delete (s, i) = $\perp$

(iv) for all $s_1, s_2, s_3, s_4 \in \underline{L}_{Set}$ and all $i \in \underline{C}_{Integer}$:

if: Is.Set(s_1) = Is.Set(s_2) = Eq.Set(s_1, s_2) = true or $s_1 = s_2 = \perp$

and Is.Set(s_3) = Is.Set(s_4) = Eq.Set(s_3,s_4) = true or $s_3 = s_4 = \perp$

then

(a) Eq.Set(Insert(s_1, i), Insert(s_2, i)) = true or Insert(s_1, i) = Insert(s_2, i) = $\perp$

(b) Eq.Set(Delete(s_1, i), Delete(s_2, i)) = true or Delete(s_1, i) = Delete(s_2, i) = $\perp$

(c) Memberof(s_1, i) = Memberof(s_2, i)

(d) Subset(s_1, s_3) = Subset(s_2, s_4).

The proof of the verification conditions for the data type *Set* of Figure V has been performed mechanically by the AFFIRM-System [25,27] and may be found in [19]. Essentially these proofs are based on the use of structural induction as discussed in Section 4.5 and on the use of the fixpoint property[10]; see also Section 7.

---

[10] i.e. the property that the minimal fixpoint is a fixpoint.

7. Properties of abstract data types and their proofs

Let $\underline{S}$, A, σ, $\underline{S}_\sigma$, A_σ be defined as in Section 4.

7.1 Formulas

Let e_1 and e_2 be (correctly typed) expressions built up from functions of the algebra A_σ and (typed) variables, each variable of type $\tau \in \underline{S}_\sigma$ ranging over the carrier set $\underline{C}_\tau$. If the values of the expressions e_1 and e_2 are of the same type, say τ, then

$$e_1 =_\tau e_2$$

or, shortly

$$e_1 = e_2$$

is called a *formula* of the algebra A_σ[11]. A *proof* of such a formula is a proof of its validity in the algebra A_σ.

In the sequel we consider properties which may be expressed as formulas. An example of such a property is

$$\text{if } q = \underline{\text{true}}$$
$$\text{then } e_1 = e_2$$

which is equivalent to the formula

$$(\underline{\text{if}}\ q\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ \bot) = (\underline{\text{if}}\ q\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ \bot)$$

---

[11] Note that in LCF [23] a formula is defined with "$\sqsubseteq$" instead of "=". Actually the use of "=" constitutes no restriction as

$$e_1 \sqsubseteq e_2$$

may be defined as

$$(\underline{\text{if}}\ e_1 \doteq e_1\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ \bot) = (\underline{\text{if}}\ e_1 \doteq e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ \bot)$$

7.2 A theorem

Let

$$e_1 = e_2 \qquad (1)$$

be a formula of the algebra A_σ. Let now e'_1 and e'_2 be the expressions which are deduced from respectively e_1 and e_2 in the following way:

- each function $F^{(A)}$ which corresponds to a user function F of the specification of σ is replaced by this user function F; similarly, $\perp_\sigma$ is replaced by $\perp_\sigma^{(L)}$;
- each variable of type σ is made to range over $\underline{L}'_\sigma$ rather than over $\underline{C}_\sigma$.

In order to prove (1) it is then sufficient to prove

$$e'_1 = e'_2 \qquad (2)$$

A proof of this theorem is by structural induction on e'_1 and e'_2; it is directly based on the definition of the functions $F^{(A)}$ and on the verification conditions. The details of the proof are left to the reader.

A proof of the formula (2) is a proof of its validity in the algebra $A_\sigma^{(L)}$ and is similar to a proof of a verification condition. Again such a proof may in most cases be performed by using structural induction and the fixpoint property; cases in which a more powerful induction principle - such as fixpoint induction or computational induction - is required, are discussed in [22].

7.3 An example

Suppose one has to prove (cf. Figure V):

$$\left.\begin{array}{l} \text{for all } t \in \underline{C}_{Set}\ ,\ i \in \underline{C}_{Integer}: \\ \qquad \text{if } t \neq \perp_{Set}\ ,\ i \neq \perp_{Integer} \\ \qquad \text{then Memberof}^{(A)}\,(\text{Delete}^{(A)}\,(t,\ i),\ i) = \underline{\text{false}} \end{array}\right\} \quad (1')$$

Taking $\sigma = \mathit{Set}$ it is sufficient to prove:

$$\left.\begin{array}{l} \text{for all } s \in \underline{L}'_{Set}\ ,\ i \in \underline{C}_{Integer}: \\ \qquad \text{if } s \neq \perp_{Set}^{(L)}\ ,\ i \neq \perp_{Integer} \\ \qquad \text{then Memberof (Delete } (s,\ i),\ i) = \underline{\text{false}} \end{array}\right\} \quad (2')$$

or, equivalently:

$$\left.\begin{array}{l} \text{for all } s \in \underline{L}_{Set}, \; i \in \underline{C}_{Integer}: \\ \qquad \text{if } Is.Set(s) = \underline{true} \text{ and } i \neq \bot \\ \qquad \text{then } Memberof\ (Delete\ (s, i), i) = \underline{false} \end{array}\right\} \quad (3')$$

The proof of (3') is by structural induction on s.

(i) *Base step*[12]: s = emptyset

Memberof (Delete (emptyset, i), i)

= Memberof (emptyset, i)

by the fixpoint property applied to Delete

= <u>false</u>

by the fixpoint property applied to Memberof

(ii) *Induction step*: s = insert (s', j)

(a) First case : $(i \doteq j) = \underline{true}$

Memberof (Delete (insert (s', j), i), i)

= Memberof (s', i)

by the fixpoint property applied to Delete

= Memberof (s', j) because $i \doteq j$

= <u>false</u>

because Is.Set(s) = <u>true</u>

i.e. Is.Set(insert(s', j)) = <u>true</u>

hence Memberof (s', j) = <u>false</u> by the fixpoint property applied to Is.Set.

(b) Second case : $(i \doteq j) = \underline{false}$

Memberof (Delete (insert (s', j), i), i)

= Memberof (insert (Delete (s', i), j), i)

by the fixpoint property applied to Delete

= Memberof (Delete (s', i), i)

by the fixpoint property applied to Memberof

= <u>false</u>

by the induction hypothesis

---

[12] Note that the base step $s = \bot^{(L)}_{Set}$ has not to be considered because of the assumption Is.Set(s) = <u>true</u>.

(c) Third case : $(i \doteq j) = \perp$

This case does not occur because

- $i \neq \perp$ by hypothesis;
- $j \neq \perp$ by the definition of $\underline{L}_{Set}$ (see Section 4.2). $\boxtimes$

8. Simultaneous fixpoint abstraction and simultaneous specifications

8.1 Simultaneous fixpoint abstraction

Simultaneous fixpoint abstraction allows to define an m-tuple $(F_1,\ldots,F_m)$ of functions, $m \geq 2$. It is expressed by writing

$$(F_1,\ldots,F_m) = [\alpha M_1.[\alpha M_2.[\ldots[\alpha M_m.(e_1, e_2,\ldots,e_m)]]\ldots] \qquad (1)$$

where $e_1,\ldots,e_m$ are expressions.

Adopting a notational convention similar to that of Section 5.4 we replace (1) by m equations

$$F_i = e'_i$$

where e'_i is deduced from e_i by replacing each M_j by F_j, $1 \leq j \leq m$, $1 \leq i \leq m$.

As an example the functions ValP and ValS in Figure VI are defined by simultaneous fixpoint abstraction.

Actually, it is convenient to systematically consider that all the functions of a specification are defined by simultaneous fixpoint abstraction; in fact, simultaneous fixpoint abstraction reduces to (normal) fixpoint abstraction whenever the condition (C1) of Section 5.1 is satisfied.

8.2 Simultaneous specifications

One may be induced to introduce data types $\sigma_1,\ldots,\sigma_m$, $m \geq 2$, the constructors of which are mutually recursive; more precisely, the specification of σ_i contains a constructor

$$\ldots : \ldots \times \sigma_{i+1} \times \ldots \rightarrow \sigma_i$$

for all i, $1 \leq i \leq m - 1$, and the specification of σ_m contains a constructor

$$\ldots : \ldots \times \sigma_1 \times \ldots \rightarrow \sigma_m$$

In that case the data types may be specified "simultaneously" - as illustrated by Figure VI. The reader should have no difficulties in generalizing the definitions of Section 5 for the case of simultbaneous specifications.

(i) General

Types : *Program, Stat*

Underlying types : *Conf, Name, Expr, Boolexpr, Bool*

(ii) Constructors

emptyprogram : → *Program*

semicolon : *Stat* x *Program* → *Program*

assign : *Name* x *Expr* → *Stat*

ifthenelse : *Boolexpr* x *Program* x *Program* → *Stat*

while : *Boolexpr* x *Program* → *Stat*

(iii) The acceptor functions

Is.Program (p : *Program*) : *Bool* = Equal-Program (p, p)

Is.Stat (s : *Stat*) : *Bool* = Equal-Stat (s, s)

(iv) The equivalence relations

Eq.Program (p_1: *Program*, p_2: *Program*) : *Bool*
= Equal-Program (p_1, p_2)

Eq.Stat (s_1: *Stat*, s_2: *Stat*) : *Bool* = Equal-Stat(s_1,s_2)

(v) External functions

ValP (p : *Program*, c : *Conf*) : *Conf*
= case
p = emptyprogram : c
p = semicolon (st, p') : ValP (p', ValS (st, c))
esac

ValS (s : *Stat*, c : *Conf*) : *Conf*
= case
s = assign (n, e) : Assign (n, e, c)
s = ifthenelse (e, p1, p2) :
if ValB (e,c) then ValP (p1,c) else ValP (p2,c)
s = while (e, p) :
if ValB (e,c) then ValS (s, ValP (p, c)) else c
esac

FIGURE VI: The simultaneous specification of the data types *Program* and *Stat*. *Program* is the data type constituted by a simple while-programming language, *Stat* represents its statements and *Conf* appropriate configurations. Note that the acceptor functions are (strict) functions with the constant value true; the equivalence relations express the syntactical equality.

9. Parameterized data types

Parameterized data types have been introduced in e. g. [29,8,28]. Essentially, a parameter ranges either over the carrier set of a data type or over the set of all types. Both cases are considered sucessively.

9.1 Data parameters

An example of such a data type is *Stack* [n : *Integer*] or, shortly, *Stack* [n] representing a stack with maximal depth n.

A specification of such a data type is a specification scheme rather than a single specification. Alternatively, adopting the notation of Section 4 one may define

$$\underline{S}_\sigma \quad = \quad \underline{S} \cup \{\mathit{Stack}\ (n) \mid n \in \underline{C}_{\text{Integer}}\} \quad ;$$

similarly, A_σ is obtained by adding for each $n \in \underline{C}_{\text{Integer}}$ a carrier set and a set of functions according to the definitions of Section 5.2.

An example (combined with a type parameter) is in Figure VII.

9.2 Type parameters

An example of such a data type is *Stack* [τ : *Type*] or, shortly, *Stack* [τ] representing a stack of elements of type τ; in this notation *Type* is to be considered as an additional key-word. Intuitively τ ranges over all types of the algebra A as well as over the type being specified; the latter feature leads for instance to the introduction of stacks the elements of which are stacks of integers.

More precisely, the set of types $\underline{S}_\sigma$ is now defined as (the smallest set satisfying):

(i) if $\rho \in \underline{S}$, then $\rho \in \underline{S}_\sigma$;

(ii) if $\rho \in \underline{S}_\sigma$, then *Stack* [$\rho$] $\in \underline{S}_\sigma$.

The algebra A_σ is defined as in Section 9.1.

An example is in Figure VII; note that if one wants to stick to the principle of strong typing it is necessary to provide also the function symbols such as Is.Stack or Push with an index $[n,\tau]$.

(i) General

Type : *Stack* [n : *Integer*, τ : *Type*]

Underlying types : *Integer*, *Bool*

(ii) Constructors

emptystack : $\rightarrow$ *Stack* [n, τ]

push : *Stack* [n, τ] x $\tau \rightarrow$ *Stack* [n, τ]

(iii) Acceptor function

Is.Stack(s : *Stack* [n, τ]) : *Bool*

= <u>if</u> Depth(s) $\leq$ n <u>then</u> <u>true</u> <u>else</u> <u>false</u>

(iv) Equivalence relation

Eq.Stack (s_1:*Stack* [n, τ], s_2:*Stack* [n, τ]) : *Bool*

= Equal-Stack (s_1, s_2)

(v) External functions

Emptystack = emptystack

Push (s : *Stack* [n, τ], e : τ) : *Stack* [n, τ]

= <u>if</u> Depth(s) < n <u>then</u> push (s, e) <u>else</u> $\perp$

Pop (s : *Stack* [n, τ]) : *Stack* [n, τ]

= <u>case</u>

s = emptystack : $\perp$

s = push (s', e) : s'

<u>esac</u>

Top (s : *Stack* [n, τ]) : τ

= <u>case</u>

s = emptystack : $\perp$

s = push (s', e) : e

<u>esac</u>

Isnew (s : *Stack* [n, τ]) : *Bool*

= <u>case</u>

s = emptystack : <u>true</u>

s = push (s', e) : <u>false</u>

<u>esac</u>

(vi) Auxiliary function

Depth (s : *Stack* [n, τ]) : *Integer*

= <u>case</u>

s = emptystack : 0

s = push (s', e): Depth (s') + 1

<u>esac</u>

<u>FIGURE VII:</u> A specification of the parameterized data type *Stack* [n : *Integer*, τ : *Type*]. Informally, n is the maximum depth of the stack, τ is the type of the elements stacked.

10. Conclusions

By defining functions algorithmically rather than axiomatically the specification method proposed avoids the problems of consistency and sufficiently-completeness. Using implicitly the constructs of LCF for the definition of new functions the method allows the specification of any data type consisting of a recursively enumerable carrier set and (partial) computable functions (cf. [22]). Using a specification to deduce a new algebra from a given one, the method solves the extension problem, and, moreover, avoids to start from "scratch". Finally, solutions to the error problem have been proposed in Section 5.3.

As shown in [21] algorithmic specifications moreover lead to a simple definition of the implementation of abstract data types. The correctness proof of such an implementation may again be formulated in terms of the algebra $A_{\sigma}^{(L)}$ and may be proved with the help of the same verification system; an example treated with AFFIRM is in [19].

In [21] it is shown that algorithmic specifications may also be used for the definition of programming languages and for the verification of their compilers.

As an additional advantage it is possible to prove algorithmic specifications "correct" by proving that they satisfy the axioms of a - not necessarily sufficiently-complete - algebraic specification of the same data type.

The price paid for these different advantages is a potential danger of "overspecification"; for instance, it is not possible to leave certain function values unspecified or to use quantifiers. Whether this shortcoming is relevant for the practice is not clear to the author. First, arbitrary choices in a function definition - such as innermost-to-outermost rather than outermost-to-innermost recursion - do not preclude different choices of the implementation: they at most make the correctness proof of the implementation more complex. Second, the striking similarities of our results with, for instance, those of [13] and the fact that is was possible to use the AFFIRM-system for proving properties of algorithmic specifications suggest that the basic ideas of the algorithmic specification method were implicitly present in several works on algebraic specifications.

Similar works are described in [16, 6]. The specification method proposed in the former paper allows the introduction of primitive recursive functions only; the latter paper has stronger similarities with the approach discussed here but seems to have not been completely worked out. By their constructive nature operational specifications [e. g. 18, 29, 26] also present analogies; as a main difference they make use of an Algol-like language rather than of a (freely chosen) term language together with the constructs of composition, λ-abstraction and fixpoint abstraction.

Further work on algorithmic specifications includes the top-down development of a real-life program and its verification, the choice and/or development of an appropriate verification system and the design of a "metalanguage" in the style of CLEAR [5].

Acknowledgments

The author is indebted for several discussions to M. Broy, C. W. Lermen, P. Raulefs, M. Wirsing and the members of the AFFIRM-group at ISI-USC.

References

[1] AUBIN R. Mechanizing structural induction, part 1. *Theoretical Computer Science* 9 (1979), 329-345.

[2] BROY M., DOSCH W., PARTSCH H., PEPPER P., WIRSING M. Existential quantifiers in abstract data types. Proc. 6th ICALP Conf. (Graz), Lect. Notes in Comp. Sc. 71 (1979), 71-87.

[3] BROY M., WIRSING M. On the algebraic extensions of abstract data types. Proc. Int. Coll. on Formalization of Programming Concepts (Peniscola), Lect. Notes in Comp. Sc. 107 (April 1981), 244-251.

[4] BURSTALL R. M. Proving properties of programs by structural induction. Comp. J. 12 (1969), 41-48.

[5] BURSTALL R. M., GOGUEN J. A. The semantics of CLEAR, a specification language. Proc. 1979 Copenhagen Winter School (Jan. 1979), Lect. Notes in Comp. Sc. 86 (1980), 292-332.

[6] CARTWRIGHT R. A constructive alternative to axiomatic data type definitions. Techn. Rep. TR 80-427, Dept. of Comp. Sc., Cornell Univ. (June 1980); also in the proceedings of the 1980 LISP Conference.

[7] EHRIG H., KREOWSKI H. J., PADAWITZ, P. Stepwise specification and implementation of abstract data types. Proc. 5th ICALP Conf. (Udine), Lect. Notes in Comp. Sc. 62 (1978) 203-226.

[8] EHRIG H., KREOWSKI H. J., THATCHER J., WAGNER E., WRIGHT J. Parameterized data types in algebraic specification languages, Proc. 7th ICALP Conf. (Noordwijkerhout),Lect. Notes in Comp. Sc. 85 (July 1980), 157-168.

[9] GOGUEN J. A., THATCHER J. W., WAGNER E. G., WRIGHT J. B. Initial algebra semantics and continuous algebras. JACM 24 (Jan. 1977), 68-95.

[10] GOGUEN J. A., THATCHER J. W., WAGNER E. G. An initial algebra approach to the specification, correctness and implementation of abstract data types. *Current Trends in Programming Methodology IV* (Yeh R., ed.), Prentice-Hall, 1978, 80-149.

[11] GOGUEN J. A. Abstract errors for abstract data types. *Formal description of programming concepts* (Neuhold E. J., ed.), North-Holland, 1978.

[12] GUTTAG J. V., HORNING J. J. The algebraic specifications of abstract data types. *Acta Informatica* 10, 1 (1978), 27-52.

[13] GUTTAG J. V., HOROWITZ E., MUSSER D. R. Abstract data types and software validation. Comm. ACM 21 (Dec. 1978), 1048-1069.

[14] GUTTAG J. V. Notes on type abstraction. IEEE Trans. on Softw. Eng. SE-6 (Jan. 1980), 13-23.

[15] KAMIN S. Final data type specifications. Proc. 7th POPL Conf. (Jan. 1980), 131-138.

[16] KLAEREN H. A simple class of algorithmic specifications for abstract software modules. Proc. 9th MFCS Conf., Lect. Notes in Comp. Sc. 88 (1980), 362-374.

[17] LEHMANN D. J., SMYTH M. B. Algebraic specifications of data types: A synthetic approach. *Math. Syst. Th.* 14 (May 1981), 97-139.

[18] LISKOV B. et al. CLU reference manual. Comp. Struct. Gr. Memo 161, Lab. Comp.-Sc., MIT (July 1978).

[19] LOECKX J. Proving properties of algorithmic specifications of abstract data types in AFFIRM. AFFIRM-Memo-29-JL, USC-ISI, Marina del Rey, 1980.

[20] LOECKX J. Algorithmic specifications of abstract data types. Proc. 8th ICALP (Acre), Lect. Notes in Comp. Sc. 115 (July 1981), 129 - 147.

[21] LOECKX J. Implementations of abstract data types and their verification. Proc. of the ECI-GI 81 Conference (Munich), Informatik-Fachberichte 50, Springer-Verlag, 96 - 108.

[22] LOECKX J. Using abstract data types for the definition of programming languages and the verification of their compilers. Techn. Rep. A 81/13 (Sept. 1981), Fachbereich 10, Univ. Saarbrücken (submitted for publication).

[23] MILNER, R. Implementation and application of Scott's logic for computable functions. Proc. ACM Conf. on Proving Assertions about Programs, SIGPLAN Notices 7 (Jan. 1972), 1 - 6.

[24] MILNER, R. Models of LCF, Stanford University Memo AIM-186 (Jan. 1973).

[25] MUSSER, D. R. Abstract data type specification in the AFFIRM System. IEEE Trans. on Softw. Eng. SE-6 (Jan. 1980), 24 - 32.

[26] STEENSGAARD-MADSEN J. A statement-oriented approach to data abstraction. *Trans. Programm. Lang. Syst.* 3 (Jan. 1981), 1 - 10.

[27] THOMPSON D. H. (Ed.). AFFIRM Reference Manual. Internal Report, USC-ISI, Marina del Rey (1979).

[28] WIRSING M. An analysis of semantic models for algebraic specifications, Intern. Summer School on Theor. Found. of Progr. Method., Munich (July 28 - Aug. 9, 1981).

[29] WULF Wm. A. (Ed.). An informal definition of ALPHARD. Techn. Rep CMU-CS-78-105, Carnegie-Mellon Univ. (Feb. 1978).

[30] ZILLES S. N. Algebraic specifications of data types. Comp. Struct. Group Memo 119, Lab. Comp. Sc., MIT (1974).